# Security Metrics for Object-Oriented Designs

Bandar Alshammari[1], Colin Fidge[2] and Diane Corney[3]
Faculty of Science and Technology
Queensland University of Technology
Brisbane, Australia.
Email: [1] b.alshammari@student.qut.edu.au [2] c.fidge@qut.edu.au [3] d.corney@qut.edu.au

***Abstract*—Several studies have developed metrics for software quality attributes of object-oriented designs such as reusability and functionality. However, metrics which measure the quality attribute of information security have received little attention. Moreover, existing security metrics measure either the system from a high level (i.e. the whole system's level) or from a low level (i.e. the program code's level). These approaches make it hard and expensive to discover and fix vulnerabilities caused by software design errors. In this work, we focus on the design of an object-oriented application and define a number of information security metrics derivable from a program's design artifacts. These metrics allow software designers to discover and fix security vulnerabilities at an early stage, and help compare the potential security of various alternative designs. In particular, we present security metrics based on composition, coupling, extensibility, inheritance, and the design size of a given object-oriented, multi-class program from the point of view of potential information flow.**

***Index Terms*—Quality; Security; Metrics; Refactoring**

## I. Introduction

Many software quality attributes have been studied and measured extensively including maintainability, performance, reusability, and reliability [1]. Security, on the other hand, has received relatively little attention. Furthermore, those security measurements which have been defined either assess security at the abstract system architecture level [2] or at the low level of program code [3]. Measuring security at the design phase, based on typical design artifacts, has not been considered even though such metrics could have efficiently eliminated software security vulnerabilities before they reach the finalised product [4] [5]. Such metrics would also allow software developers to compare the security level of various alternative designs under consideration.

This paper proposes a new set of metrics which are capable of assessing information-flow security of object-oriented designs. In previous work, we developed seven security metrics for assessing the security of a single object-oriented class [6]. These metrics measured Data Encapsulation and Cohesion for a given class. Here we extend that work to consider entire class hierarchies.

We define our metrics based on the quality properties for object-oriented programs specified by Bansiya and Davis [1], using five properties which are related to the overall design of an object-oriented program: composition, coupling, extensibility, inheritance, and design size. Our metrics aim to measure any potential information flow which could occur between objects instantiated from the design's classes.

However, in order to measure the impact of these properties on information flow, we need security-annotated class diagrams. In our case, we use UMLsec and SPARK's annotations to identify confidential data [7] and to express the information flow relations between attributes, methods, and classes of a given design [8]. Once the metrics' results are identified for a number of alternative designs, it is then easy to choose the most secure design.

## II. Related Work and Research Problem

Most current studies on software security admit that there is no such thing as a completely secure program, but there are nevertheless various ways of reducing security risks and vulnerabilities [9] [10]. A common approach to developing secure programs is to enforce security at the implementation stage. Several projects have investigated information flow through computer program code, via type analysis [11], data/control-flow analysis [12], and ways of identifying and eliminating program code vulnerabilities [10] [13].

Chowdhury et al. [3] defined a number of security metrics that assess the security of a given program based on code inspections. However, their metrics require full system implementations to assess security which makes it impossible to fix problems at design time.

Instead, the most efficient approach is to enforce security at early phases of the software development lifecycle such as during the design phase [4] [5]. The National Institute of Standards and Technology [14] stated that eliminating vulnerabilities in the design stage can cost 30 times less than fixing them at a later stage. One of the earliest studies in this area was the development of software security design principles by Saltzer and Schroeder [15]. Additionally, Bishop's [16] and Viega and McGraw's [17] texts identify several significant security design principles. However, these principles are intended as guidance to help develop secure systems, mainly operating systems, and are not capable of quantifying the security levels of designs. Thus, there is a need for security metrics which objectively measure the security of a given program directly from its design artifacts.

A study that defined design metrics which measure certain software quality attributes was conducted by Bansiya and Davis [1]. They identified a Quality Model for Object-Oriented Design (QMOOD) [1]. Their approach aims to measure the

quality of various object-oriented design attributes such as reusability, flexibility, and functionality based on their relevance to certain quality design properties (e.g. abstraction, cohesion, and coupling). Even though the study covered most design quality attributes, it did not consider security.

In general, studies which focus on 'programming in the small' [18], i.e., at the level of individual program statements, are not the most efficient approach to achieving software security. In this paper, we instead extend our previously defined design metrics which evaluate security based on a single class diagram [6] to entire class hierarchies. We define several new security metrics for a complete object oriented design. They can be used to compare different designs for the same program and identify the most secure one.

## III. Assumptions and Annotations

We have developed our security design metrics to be capable of quantifying the security level of a given object-oriented program based on its design quality properties (composition, coupling, extensibility, inheritance, and design size). These metrics are meant to be comparative rather than absolute measures. They can be used to compare various alternative designs for the same program with respect to their security properties. They are different from typical "code complexity" metrics, which measure explicit syntactic properties of the code such as the number of variables and lines of code. Instead, our metrics measure (usually implicit) potential information flow properties within a program design. They have been scaled to all fit within the range 0 to 1 with lower values considered desirable. The metrics can show which designs are more secure by choosing the lowest values.

To apply our metrics to a given design, we assume that system designers will provide annotated UML class diagrams using UMLsec and SPARK's annotations. UMLsec [7] is an extension of the Unified Modeling Language which labels objects as 'critical' if they contain data which can be of a security risk at any point. It also associates a 'secrecy' tag with data which needs to be kept confidential [7].

On the other hand, SPARK is a programming language for security-critical code in which the programmer may annotate subroutines with the intended data flow between variables and parameters. The SPARK compiler then performs a data-flow analysis to confirm that the code does indeed have the characteristics the programmer intended. SPARK's annotations consist of a "derives x from Y" block which explains how the value of a certain variable or return value x is potentially derived from the value of other method parameters or variables Y [8]. Our metrics assume the "derives from" annotations identify the flow of data between methods and classes.

To have consistent designs, we assume that a class is labeled "critical" if and only if it contains classified attributes labeled as "secrecy" or if it has an attribute which derives its value from a classified attribute. Similarly, an attribute which derives its value from a classified attribute or method parameter must be labeled as "secrecy".

## IV. Metrics Development and Definitions

We developed our security metrics for object oriented designs based on the analysis of software quality design properties defined in the Quality Model for Object-Oriented Design [1]. These properties include: composition, coupling, extensibility, inheritance, and design size. We studied each property and its relevance to designing secure software to define our metrics so that if any security-critical attributes have changed in a design then the relevant metrics will reflect this change as well.

### A. Composition

Aggregation and composition are extensively mentioned in the context of object-oriented programming. Aggregation is an association between two or more system components/objects [19]. It's been described as a 'whole-part association', where one object is the whole and the others are the parts [19]. However, composition is a stronger type of aggregation which has a 'lifetime dependency' between composite objects and the whole object [19]. Therefore, deleting the whole object would result in deleting the composite ones [19]. In object-oriented programming, composition is implemented through the use of inner/nested classes [20], by implementing the composite whole class as the outer class and the composite part class as the inner one. It is expected that no other class would have access to the inner class directly, therefore, access to inner classes is done by first having access to the outer class [21].

However, this is not always true in program code since some programming languages treat inner classes as independent ones, and therefore allow direct access to them [21]. However, at the design level, we assume this can't occur and that the design describes the intended implementation accurately. Such implementation-dependent issues are beyond the scope of design-time analyses.

Composition yields a weak possibility of potential information flow for classified data when considering information security. This risk has been identified in the field of software security, and as a result it is recommended to avoid using inner classes security-critical code [21]. However, in our case we assume that using inner classes is secure since most programming languages don't allow external access to inner classes, unless they are marked as public.

Figure 1 shows a class hierarchy which has deployed composition. The design includes two classified data attributes, $\delta$ of type U and $\gamma$ of type V, distributed among two classes: class A (outer class) and class B (inner class). (We use Greek letters in these examples to denote classified data.) Our composition metric aims to increase the use of critical composite parts which contain classified data similar to class B and limit the use of composite whole classes similar to class A. This will reduce potential information flow of classified data from composite-whole to composite part classes.

*1) Composite-Part Critical Classes (CPCC):* This metric is proportional to "The ratio of the number of critical composed-part classes to the total number of critical classes in a design".

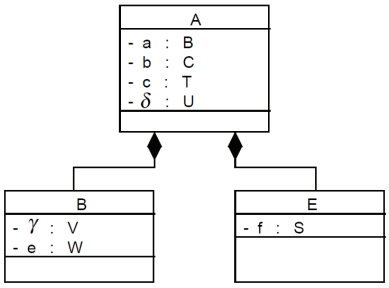


Figure 1. Composition Class Hierarchy

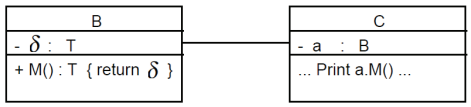


Figure 2. Coupling through Methods

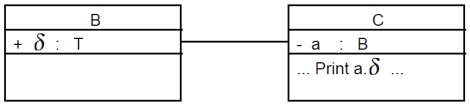


Figure 3. Coupling through Attributes

It aims to reward the use of inner classes for holding classified data, and penalise the use of outer classes for this purpose. We assume that in order to access inner classes, it is essential to access the outer class, and therefore no direct access can be made to the inner class [21]. From this point of view, storing private classified data in inner classes is a more secure solution since this data has less chance to be exposed to the public. Increasing the number of critical composed-part classes in a design gives a lower value of this metric, and hence indicates a more secure design.

Consider a set of critical classes in a design $D$ as $CC = \{cc_1, ..., cc_n\}$ and the composed-part critical classes in the same design as $CP = \{cp_1, ..., cp_n\}$ such that $CP \subseteq CC$. Then, we define the Composite-Part Critical Classes metric as follows, where $|S|$ is the size of set $S$.

$$CPCC(D) = 1 - \left(\frac{|CP|}{|CC|}\right)$$

### B. Coupling

Coupling is one of the most important software design properties. It is defined as the interaction degree an object has with other objects, measured by the number of links it has with these objects [19]. A number of metrics have been defined to measure the coupling degree between classes of an object-oriented program [22] [23]. A design time coupling metric is defined by Bansiya and Davis [1] called Direct Class Coupling (DCC). Their metric measures the number of other classes that a certain class interacts with [1]. A system with low coupling is considered a better design with regard to reusability, understandability, and extensibility [1] [19].

The impact of coupling on security has been discussed by Liu and Traore [24] in their study which shows a strong correlation between coupling and a system's attackability. Systems with high coupling are a greater target for successful attacks unlike systems with low coupling [24]. With regard to security coupling metrics, a study conducted by Chowdhury et al. [3] measures the security coupling between a program's methods based on code inspections, but our security coupling metric can be applied based on the design of a given program only.

*1) Critical Classes Coupling (CCC):* This metric aims to find the degree of coupling between classes and classified attributes in a given design. It is defined as "The ratio of the number of all classes' links with classified attributes to the total number of possible links with classified attributes in a given design". It is calculated based on the theory of directed weighted links. In our context, a directed link shows the classes which may reference or access certain classified attributes in other classes. For each class that may access a classified attribute (as indicated by one or more 'derives from' annotations), we add a weight of one. To find the ratio of the coupling, we need to first identify the sum of the classified attributes' accessing or referencing weights. Then, we divide this number by the total number of links to all classified attributes, which is the total number of classes, less the classified attribute's own class, times the total number of classified attributes.

This metric aims to penalise programs with high coupling. Therefore, it produces lower values for fewer interactions between classes and classified attributes, and hence a lower chance of potential flow of classified data.

Figure 2 illustrates the coupling we measure by this metric, through the calling of methods in other classes which return classified data. In this case, objects of class C call method M in class B which returns the value of classified attribute $\delta$. However, coupling through the use of public classified attributes as in Figure 3 is not covered by this metric. This case is measured by our single class metrics [6].

Consider a set of classes in a design $D$ as $C_i, i \in \{1, \ldots, c\}$ and a set of classified attributes as $CA_j, j \in \{1, \ldots, ca\}$, and let $\alpha(CA_j)$ be the number of classes which may interact with classified attribute $CA_j$. Then, the CCC metric for design D can be expressed as follows.

$$CCC(D) = \frac{\sum_{j=1}^{ca} \alpha(CA_j)}{(|C| - 1) \times |CA|}$$

### C. Extensibility

Extensibility is the property which allows a certain class or method to be extended by other classes or methods [19]. McGraw and Felten's text [21] identifies twelve rules for developing more secure Java code. One of these rules is the necessity of preventing classes and methods from being extended. The text mentions that extensibility is an enemy of secure code, and therefore it is essential to make classes and methods inextensible (finalised) unless there is a convincing reason not to do so [21]. With regard to this security coding

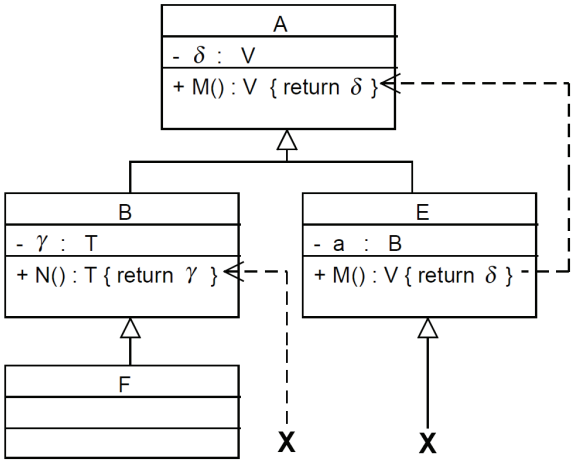


Figure 4. Class Hierarchy with Extensibility

rule, we have identified two metrics which measure how it is possible to extend the system's classes, methods, and attributes from a design diagram perspective. The metrics are divided into two kinds: one measures the extensibility of critical classes while the other measures the extensibility of classified methods.

Figure 4 illustrates the concept of extensibility. It shows that class E and method N in class B are not extensible. However, other classes and methods could be extended such as class B and method M in class A.

*1) Critical Classes Extensibility (CCE):* This metric is defined as "The ratio of the number of the non-finalised critical classes in a design to the total number of critical classes in that design". Since extensible critical classes could allow other classes to have access to their classified data, their presence increases the 'attack surface' size [25]. Making critical classes inextensible eliminates this risk, and hence reduces the possibility of information flow from these classes, so the CCE metric penalises designs with extendible critical classes, such as classes A and B in Figure 4. Considering the design in Figure 4, the CCE metric would give us a value of two non-finalised critical classes, i.e. class A and B, over three critical classes in that design.

Consider a set of classes in a design $D$ as $CC = \{cc_1, \ldots, cc_n\}$ and extensible critical classes in the same design as $ECC = \{ecc_1, \ldots, ecc_n\}$ such that $ECC \subseteq CC$. Then, we define the Critical Classes Extensibility metric as follows.

$$CCE(D) = \frac{|ECC|}{|CC|}$$

*2) Classified Methods Extensibility (CME):* This metric measures the proportion of non-finalised classified methods which can be extended against the total number of classified methods in a design. We define it as "The ratio of the number of the non-finalised classified methods in a design to the total number of classified methods in that design". Non-finalised methods allow other methods in other classes to override them. This is a security risk if such methods are classified. This can be avoided by declaring classified methods as inextensible (final), and hence the risk of losing control over classified data can be reduced. Therefore, this metric rewards designs with inextensible classified methods, similar to method N in class B from Figure 4.

To apply the CME metric to Figure 4, we get two non-finalised classified methods, i.e. method M in classes A and B respectively over three classified methods in the whole design which means two thirds of the classified methods in the design are not finalised.

Consider a set of classified methods in a design $D$ as $CM = \{cm_1, \ldots, cm_n\}$ and extensible classified methods in the same design as $ECM = \{ecm_1, \ldots, ecm_n\}$ such that $ECM \subseteq CM$. Then, we define the Classified Methods Extensibility metric as follows.

$$CME(D) = \frac{|ECM|}{|CM|}$$

### D. Inheritance

Inheritance is an object-oriented mechanism which allows programmers to provide classes with generalisation and specialisation relationships [19]. Subclasses in an inheritance hierarchy automatically obtain features of their superclasses [19]. The work of Lorenz and Kidd [26] in 1994 aimed to measure inheritance via the number of inherited methods, number of overridden methods, and number of new methods in a class [26]. Bansiya and Davis [1] defined inheritance as the degree of reuse, measured by finding the ratio of the number of methods inherited by a class to the total number of methods accessed by member methods of the class.

Both of these sets of metrics can be measured from a design perspective. However, measuring how inheritance could have an impact on security has not been discussed previously. From the security point of view, inheritance could allow subclasses to acquire privileges over classified data in superclasses, which increases the chance of potential classified information flow. As a result, we have grouped our metrics into four types which cover various kinds of information flow in an inhertiance hierarchy.

We illustrate these metrics with regard to Figure 5 which shows a class diagram with inherited classified data. Classes A, B, and E are superclasses, and therefore their public and protected attributes and methods, such as protected method N in class B, can be inherited by their subclasses, i.e. classes F, G, and I. However, attributes declared as private in a superclass cannot be inherited such as classified attributes $\delta$ and $\gamma$ in classes A and B respectively.

*1) Critical Superclasses Proportion (CSP):* This inheritance metric measures the proportion of critical superclasses in an inheritance hierarchy. It is defined as "The ratio of the number of critical superclasses to the total number of critical classes in an inheritance hierarchy". The metric identifies the proportion of critical classes that contain classified data which could be accessed by subclasses. It rewards a lower overall proportion of critical superclasses in a design, and penalises use of critical superclasses. In Figure 5 there are two critical

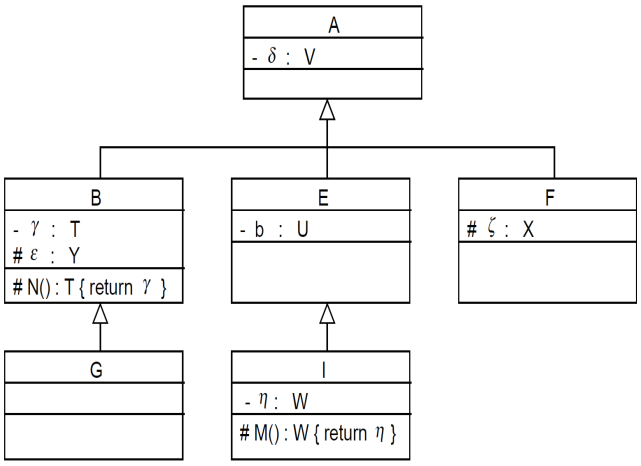


Figure 5. Class Hierarchy with Inheritance

superclasses, i.e. classes A and B, compared to four critical classes in the whole hierarchy.

Consider a set of critical classes in hierarchy $H$ as $CC = \{cc_1, \ldots, cc_n\}$ and critical superclasses in the same hierarchy as $CSC = \{csc_1, \ldots, csc_n\}$ such that $CSC \subseteq CC$. Then, we define the Critical Superclasses Proportion as follows.

$$CSP(H) = \frac{|CSC|}{|CC|}$$

*2) Critical Superclasses Inheritance (CSI):* This metric is defined as "The ratio of the sum of classes which may inherit from each critical superclass to the number of possible inheritances from all critical classes in a class hierarchy". The number of possible inheritances from critical classes is the product of the number of classes less one times the number of critical classes in the class hierarchy. This metric penalises class hierarchies in which critical classes appear near the top, and rewards those in which critical classes appear near the bottom. Lower values of this metric indicate that fewer classes may inherit from each critical superclass, and hence there is a less chance of information flow to subclasses. This also means that critical superclasses are towards the bottom of the hierarchy, so these classes are easier to secure [25] because they can have fewer inhertiting subclasses. In Figure 5, the classes which may inherit from critical superclasses A and B are classes B, E, F, and G. If we apply the CSI metric to Figure 5, the value of this metric is calculated as follows. The number of classes which may inherit from each critical superclass is six, i.e. 5 can inherit from class A and 1 can inherit from class B. The total number of possible inheritances from all critical classes in Figure 5 is twenty, i.e. for each of the 4 critical classes, there are 5 other classes that could potentially inherit from it.

Consider a set of classes in hierarchy $H$ as $C_i, i \in \{1, \ldots, c\}$, a set of critical classes in the same hierarchy as $CC_j, j \in \{1, \ldots, cc\}$, and a set of critical superclasses in the same hierarchy as $CSC_k, k \in \{1, \ldots, csc\}$ where $CSC \subseteq CC$ and $CC \subseteq C$. Let $\beta(CSC_k)$ be be the number of classes which may inherit from the critical superclass $CSC_k$. Then, the Critical Superclasses Inheritance metric for hierarchy $H$ is defined as follows.

$$CSI(H) = \frac{\sum_{k=1}^{csc} \beta(CSC_k)}{(|C|-1) \times |CC|}$$

*3) Classified Methods Inheritance (CMI):* This metric is defined as "The ratio of the number of classified methods which can be inherited in a hierarchy to the total number of classified methods in that hierarchy". It measures the proportion of classified methods which are exposed to inheritance by other classes. Hierarchies with lower values of CMI have fewer classified methods that are exposed to inheritance, and thus represent a more secure system. Figure 5 shows a classified method N in class B which is inheritable. The CMI metric measures this type of method and penalises their use in a multi-class design. By contrast, Figure 5 has another inheritable classified method M in class I. However, this method is not counted by this metric since the enclosing class I is not a superclass.

Consider a set of classified methods in hierarchy $H$ as $CM = \{cm_1, \ldots, cm_n\}$ and the classified methods which could be inherited in the same hierarchy as $MI = \{mi_1, \ldots, mi_n\}$ such that $MI \subseteq CM$. Then, we define the Classified Methods Inheritance metric as follows.

$$CMI(H) = \frac{|MI|}{|CM|}$$

*4) Classified Attributes Inheritance (CAI):* We define this metric as "The ratio of the number of classified attributes which can be inherited in a hierarchy to the total number of classified attributes in that hierarchy". It measures the proportion of classified attributes which are exposed to inheritance by other classes. Similar to the CMI metric, CAI aims to show that hierarchies with lower values of CAI have fewer classified attributes exposed to inheritance, and thus produce a more secure system. This metric can be illustrated using the classified attributes in Figure 5. Attribute $\varepsilon$ in B is inheritable and is counted by this metric. However, attribute $\zeta$ in class F is not inheritable in this design since class F is not a superclass. Therefore, it is not counted.

Consider a set of classified attributes in hierarchy $H$ as $CA = \{ca_1, \ldots, ca_n\}$ and the classified attributes which could be inherited in the same hierarchy as $AI = \{ai_1, \ldots, ai_n\}$ such that $AI \subseteq CA$. Then, we define the Classified Attributes Inheritance metric as follows.

$$CAI(H) = \frac{|AI|}{|CA|}$$

### *E. Design Size*

Design size measures the number of classes in a design [1]. A metric for measuring design size in object-oriented designs defined by Bansiya and Davis [1] is called Design Size in Classes (DSC). DSC is a count of the total number of classes in a certain design [1]. Bansiya and Davis' study also revealed that design size has a major impact on a program's reusability and functionality [1]. With respect to security, the

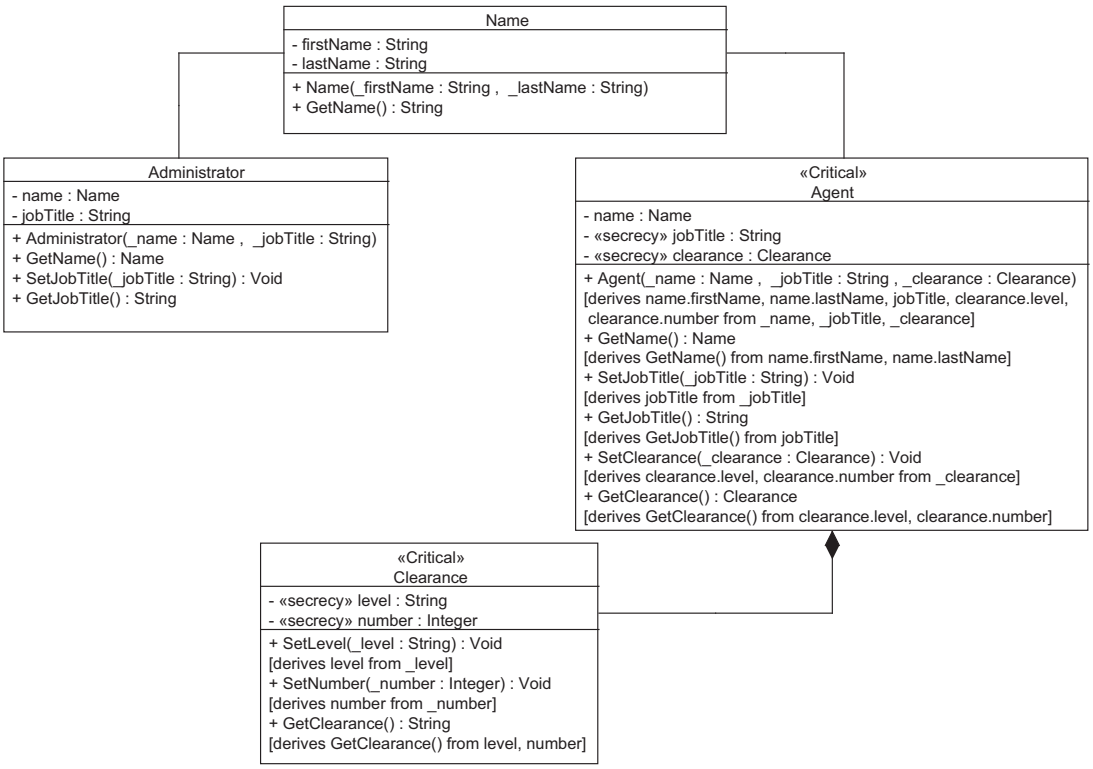


Figure 6. Defence System Original Class Hierarchy

size of object-oriented designs, to our knowledge, has not been utilised. Nevertheless, Chowdhury et al.'s study [3] defines a metric which measures the ratio of critical elements in a specific program's code. This metric requires the system to be fully implemented in order to calculate such a ratio.

*1) Critical Design Proportion (CDP):* This metric measures the impact of the size of a certain design on security. We define it as "The ratio of number of critical classes to the total number of classes in a design". It measures the proportion of classes which store classified data, i.e. critical classes. A higher proportion of critical classes in a design indicates higher security risks for potential information flow. Designs with lower values of CDP indicate fewer critical classes compared to other designs of the same size, and hence more secure systems [16].

Consider a set of classes in design $D$ as $C = \{c_1, \ldots, c_n\}$ and the critical classes in the same design as $CC = \{cc_1, \ldots, cc_n\}$ such that $CC \subseteq C$. Then, we define the Critical Design Proportion metric as follows.

$$CDP(D) = \frac{|CC|}{|C|}$$

## V. Metrics Case Study

The following case study illustrates how our software security design metrics are used. They can be applied once a complete UML class diagram, or similar, is constructed for a given system. Like our single class security metrics case study [6], this class diagram must include UMLsec and SPARK's annotations in addition to the standard elements of a class diagram.

### A. Annotated UML Class Diagrams

This section shows an annotated class diagram for a planned computer program for the Department of Defence. The class diagram in Figure 6 has been annotated using UMLsec and SPARK's annotations. The UMLsec annotations show the data which needs to be kept confidential while SPARK's annotations identify how data flows between the program's classes, attributes, and methods. The defence system class is responsible for storing information about a person working within the Department of Defence. This person can be either an agent or an administrator. The Agent class is responsible for storing an agent's information. This includes the agent's name, job title, security clearance level and number. We assume that the security clearance level and number are codes which describe the security classification for a given agent. Details of an agent's job title and security clearance are meant to be kept secret.

The Administrator class is responsible for storing information about an administrator. This information consists of the administrator's name and job title. Unlike the Agent class, the Administrator class does not contain any confidential data.

Both the Agent and Administrator classes use the same Name class.

### B. Refactored UML Class Diagrams

In this section we illustrate the capabilities of our metrics by applying them first to the original design of the defence system, and then to three refactored versions of the original design using one or more of the refactoring rules defined by Fowler [27]. (These additional designs omit the "derives from" annotations for brevity.)

For instance, Figure 7 shows a design which has been constructed after applying a number of refactoring steps to the original one. It differs from the original design in the number of classes. The system now has three classes instead of four, thanks to the refactoring rule Inline Class which has been applied to the original design through the merging of the Clearance and Agent classes. All the attributes and methods of the Clearance class have been moved to the Agent class by applying this rule.

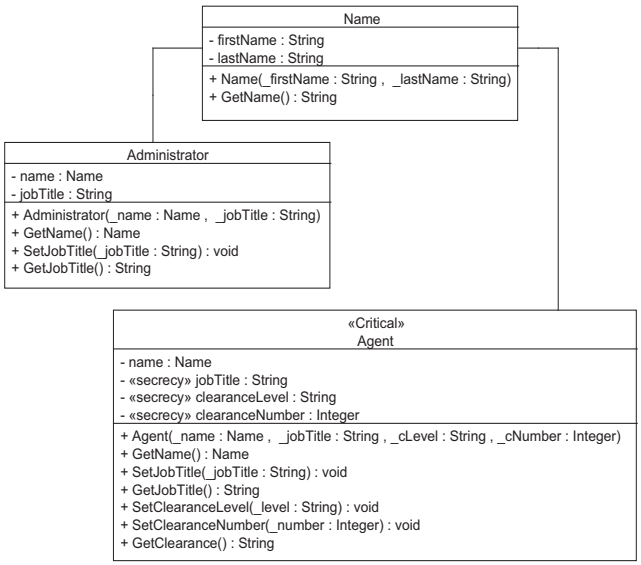


Figure 7. Defence 2 Class Hierarchy

Another refactored design is shown in Figure 8. It keeps the Clearance class but since the Agent and Administrator classes have similar features, it has used inheritance. To do this we applied the following refactoring rules: Extract Superclass, Pull up Field, and Pull up Method. Extract Superclass was used to combine similar attributes and methods in the Agent and Administrator classes by creating a superclass called Staff. The Pull up Field and Pull up Method refactoring rules were used to move the same fields and methods from both classes to the superclass. These fields are name and jobTitle, while the methods are GetName, SetJobTitle, and GetJobTitle. The only exception is keeping the Administrator job title field in its original class since it is not confidential and it can be exposed to the public.

Figure 9 also used the Extract Superclass refactoring rule to separate a new class from the Administrator and Agent classes called Staff. Then it has used the Inline Class refactoring rule to combine the Clearance and Agent classes. It then used the Inline Class refactoring rule to combine the Staff and Name classes. Figure 9 has also labelled the critical class Agent, which contains classified attributes and methods, as 'final' to indicate that this class can't be extended.

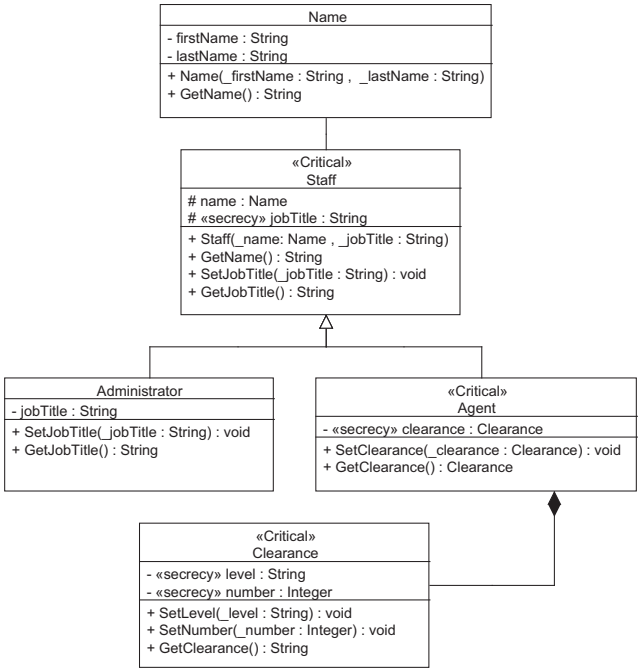


Figure 8. Defence 3 Class Hierarchy

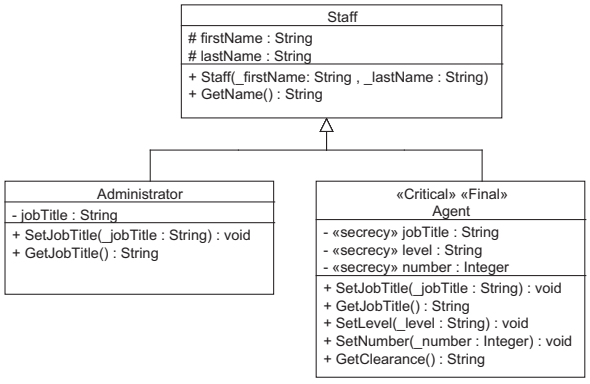


Figure 9. Defence 4 Class Hierarchy

### C. Security Metrics Results

Table 1 shows the results of applying our metrics to the four designs shown in Figures 6 to 9. For instance, to calculate the CPCC metric for Designs 1 and 2, we have to find

Table I
SOFTWARE SECURITY METRICS RESULTS

| Metric Name | Defence 1 | Defence 2 | Defence 3 | Defence 4 |
|---|---|---|---|---|
| **CPCC** | 0.50 | 1 | 0.67 | 1 |
| **CCC** | 0.17 | 0 | 0.125 | 0 |
| **CCE** | 1 | 1 | 1 | 0 |
| **CME** | 1 | 1 | 1 | 0 |
| **CSP** | 0 | 0 | 0.50 | 0 |
| **CSI** | 0 | 0 | 0.50 | 0 |
| **CMI** | 0 | 0 | 0.60 | 0 |
| **CAI** | 0 | 0 | 0.50 | 0 |
| **CDP** | 0.50 | 0.33 | 0.60 | 0.33 |

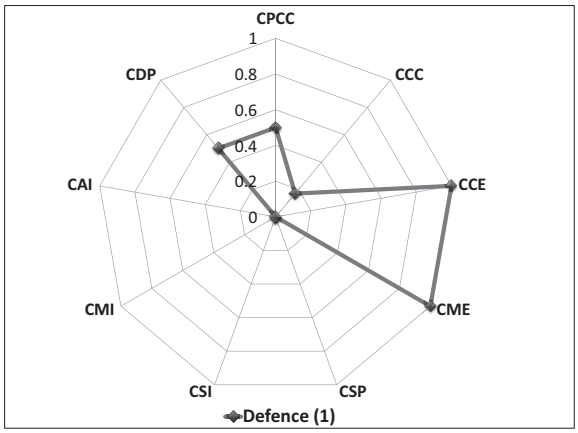


Figure 10. Metrics for Defence 1

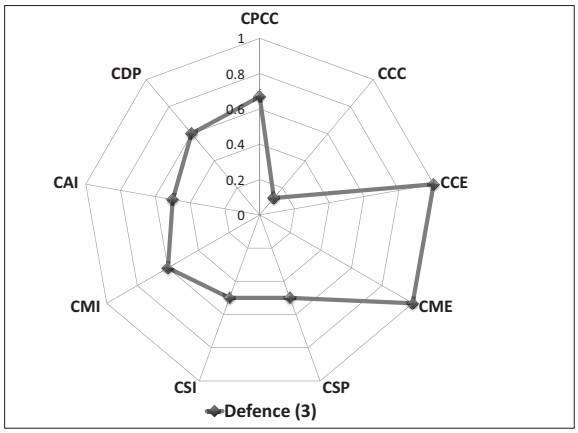


Figure 12. Metrics for Defence 3

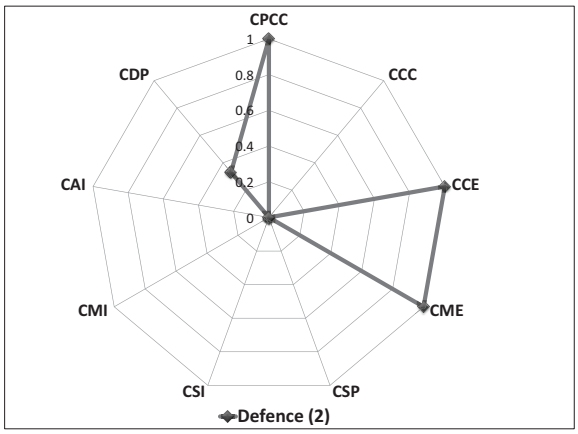


Figure 11. Metrics for Defence 2

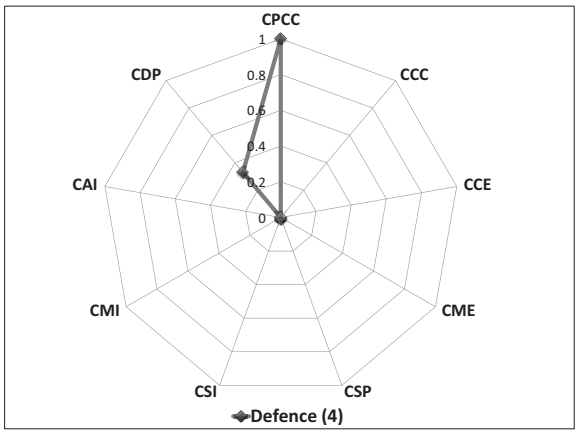


Figure 13. Metrics for Defence 4

out the number of critical composed-part classes and the total number of critical classes in both designs. This is one critical-composed part class for Design 1 while Design 2 has none. In addition, there are two critical classes in Design 1 while Design 2 has only one. According to the definition of the CPCC metric, we have to divide the number of critical composed-part classes by the total number of critical classes for each design to get its result.

The CCC metric counts the number of interactions with each classified attribute and divides it by the maximum number of possible interactions with these attributes. This number can be calculated by multiplying the number of the design's classes less one by the number of classified attributes. In the case of Design 3, which has four classified attributes, the actual number of interactions with these attributes is two while the number of possible interactions is sixteen. Dividing these two numbers gives us 0.125 in this case.

Additionally, Design 3 has the highest ratio of critical classes compared to other designs, so this design has the highest value of the CDP metric which is calculated by diving the number of critical classes in a design by the total number of classes in that design.

To calculate the CSP metric for Design 3, we first need the number of critical superclasses in all of the inheritance hierarchy, which is one. Then, this number is divided by the number of critical classes in the hierarchies, which is three.

The CSI metric is calculated by first counting the number of classes which may inherit each critical superclass, which is two in Design 3. Then, we divide this number by the maximum number of classes which could possibly inherit from these critical superclasses, which is four in Design 3.

The CMI metric is calculated by counting the number of inheritable classified methods in all of the inheritance hierarchy, which is three in Design 3 divided by the number of classified methods in that hierarchy, which is five.

The CAI metric is computed by dividing the number of inheritable classified attributes in all of the inheritance hierarchy, which is one in Design 3 by the number of classified attributes in that hierarchy, which is two.

The CCE metric is calculated by counting the number of extensible critical classes, which is zero in Design 4, and then dividing that by the total number of critical classes.

Finally, the CME metric is calculated by counting the number of extensible classified methods, which is also zero in Design 4 and then diving that by the total number of classified methods. If the class is labelled as 'final' this means that it is not extensible and also means that all of its methods are final and not extensible even though they are not labelled as final.

An easy way of comparing the results of these metrics is to show them on radar charts (Figures 10 to 13). Given that our metrics are designed so that lower values are considered more secure, graphs which are closer to the centre of the charts indicate greater security. It can be seen that Design 4 shows the lowest values for all metrics compared to the other designs except for the CPCC metric, so we could say that this design has yielded the most secure design for all properties except composition. Designs which use composition in the same way have shown the same results. Designs 2 and 4 show the highest values for the CPCC metric. Therefore, these designs are the most insecure designs for composition. However, Design 1 is the most secure design with regard to the CPCC metric since it has the lowest ratio of critical composed-part classes.

With regard to the design property of coupling, Design 1 shows the least secure design since it has the highest value among the other designs for the CCC metric. This is because it has the highest number of links with classified attributes. By contrast, Design 2 and 4 are the most secure with regard to coupling. This is because Design 2 doesn't have any inter-class references or links with classified attributes while Design 4 has replaced the traditional association between classes with inheritance.

With regard to extensibility, the only design which has declared all of its critical classes as 'final' is Design 4, which finalises all of its methods. This has resulted in this design being the most secure with regard to both the CCE and CME metrics. The other designs have yielded identical results since none have finalised any of their critical classes or classified methods.

With regard to inheritance, Design 4 has shown the lowest values for all of the inheritance metrics similarly to Designs 1 and 2 which have not used inheritance. Design 4 has used inheritance in a way such that there exists neither critical superclasses nor classified attributes or methods which could be inherited. Therefore, Designs 1, 2 and 4 are the most secure designs in this regard with a preference for Design 4 over the others since it has used inheritance in a secure way. On the other hand, Design 3 has shown the highest results, and hence the least secure design, in terms of all inhertiance metrics.

With regard to the design size property, Designs 2 and 4 have the lowest ratio of critical classes compared to the total number of classes as the CDP metric shows. Thus, they are the most secure designs for this property. On the other hand, Design 3 is the least secure design for this property since its CDP metric reveals the highest ratio of critical classes in this design.

## VI. Conclusion and Future Work

In this work, we have defined a suite of security metrics applicable to the entire design of a multi-class object-oriented system. This suite has covered a number of software design quality properties consisting of: composition, coupling, extensibility, inheritance, and design size. They provide software designers with a simple approach for identifying where security vulnerabilities might occur from the perspective of information flow of confidential data, and thus with the ability to compare the security of various designs.

In principle, the metrics could be calculated automatically by a UML tool once the design is augmented with UMLsec's annotations (which identify the confidential data) and SPARK[1]s annotations (which show how data flows within the program). Instead, however, we are developing a tool to automatically calculate these metrics from program code, to

allow us to assess their usefulness on existing large-scale systems. This approach has the advantage of allowing the "derives from" relationships to be determined automatically.

## ACKNOWLEDGMENTS

We wish to thank the anonymous reviewers for their many helpful comments. Alshammari gratefully acknowledges the Ministry of Higher Education in Saudi Arabia for sponsoring his PhD studies. Fidge and Corney's contribution to this research was funded in part by the Australian Research Council and the Defence Signals Directorate through ARC-LP grant LP0776344.

## REFERENCES

[1] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, pp. 4–17, 2002.

[2] P. K. Manadhata, K. M. C. Tan, R. A. Maxion, and J. M. Wing, "An approach to measuring a system's attack surface," Pittsburgh, PA, Tech. Rep., 2007.

[3] C. B. Chowdhury, Istehad and M. Zulkernine, "Security metrics for source code structures," in *Proceedings of the Fourth International Workshop on Software Engineering For Secure Systems*. Leipzig, Germany: ACM, 2008, pp. 57–64.

[4] A. Sachitano, R. O. Chapman, and J. A. Hamilton, "Security in software architecture: a case study," in *Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop*, 2004, pp. 370–376.

[5] E. A. Schneider, "Security architecture-based system design," in *Proceedings of the 1999 workshop on New Security Paradigms*. Ontario, Canada: ACM, 2000, pp. 25–31.

[6] B. Alshammari, C. J. Fidge, and D. Corney, "Security metrics for object-oriented class designs," in *Proceedings of the Ninth International Conference on Quality Software (QSIC 2009)*. Jeju, Korea: IEEE, 2009, pp. 11–20.

[7] J. Jürjens, *Secure systems development with UML*. Berlin, Germany: Springer, 2005.

[8] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. London, Great Britain: Addison-Wesley, 2003.

[9] G. McGraw, *Software Security: Building Security In*. Upper Saddle River, NJ: Addison-Wesley, 2006.

[10] M. Howard and D. LeBlanc, *Writing Secure Code*. Redmond, Wash.: Microsoft Press, 2002.

[11] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, pp. 167–187, 1996.

[12] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 5–19, 2003.

[13] K. Maruyama, "Secure refactoring - improving the security level of existing code," in *Proceedings of the Second International Conference on Software and Data Technologies (ICSOFT 2007)*, Barcelona, Spain, 2007, pp. 222–229.

[14] NIST, "The economic impacts of inadequate infrastructure for software testing," Gaithersburg, MD, Tech. Rep., 2002.

[15] J. H. Saltzer and M. D. Schroeder, "The protection of information in operating systems," in *Proceedings of the IEEE*, vol. 63, no. 9, 1975, pp. 1278–1308.

[16] M. Bishop, *Computer Security: Art and Science*. Boston: Addison-Wesley, 2003.

[17] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston: Addison-Wesley, 2002.

[18] R. C. Seacord, *Secure Coding in C and C++*. Upper Saddle River, NJ: Addison-Wesley, 2006.

[19] S. Bennett, S. McRobb, and R. Farmer, *Object-Oriented Systems Analysis and Design Using UML*, 3rd ed. Maidenhead: McGraw-Hill Higher Education, 2006.

[20] R. Simmons, *Hardcore Java*, 1st ed. Sebastopol, CA: O'Reilly, 2004.

[21] G. McGraw and E. Felten, *Securing Java: Getting Down to Business with Mobile Code*, 2nd ed. New York: Wiley Computer Pub., 1999.

[22] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, pp. 476–493, 1994.

[23] D. J. W. Briand, Lionel C. and J. K. Wust, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 25, pp. 91–121, 1999.

[24] M. Y. Liu and I. Traore, "Empirical relation between coupling and attackability in software systems: a case study on DOS," in *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security Ottawa*. Ottawa, Ontario, Canada: ACM, 2006, pp. 57–64.

[25] M. Howard, "Attack surface: Mitigate security risks by minimizing the code you expose to untrusted users," *Microsoft MSDN Magazine*, vol. 11, 2004.

[26] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics: A Practical Guide*. Englewood Cliffs, NJ: PTR Prentice Hall, 1994.

[27] M. Fowler, *Refactoring: Improving The Design of Existing Code*. Reading, MA: Addison-Wesley, 1999.